Program Restructuring

Ralph E. Johnson
University of Illinois at Urbana-Champaign
Johnson@cs.uiuc.edu

Abstract:

Program restructuring is an activity that has been traditionally applied to legacy software, but recently it has been advocated during the production of new systems. When carried out properly, it increases the value of legacy software and improves the quality of new software. However, it is not practiced very much. Part of the problem is the lack of tool support, but a bigger problem is lack of understanding on the part of software developers. If program restructuring is properly supported, it can become widely used. If it is widely used, it will improve the quality and decrease the cost of software.

Program Restructuring
Ralph E. Johnson

Program restructuring is usually thought of as an activity for legacy software. However, it can also be used to develop new software, and is invaluable for software that is only a few years old but is being actively extended. Some software development processes require constant rewriting of the software. In the past, we have tried to avoid rewriting software, but instead we should try to make rewriting software easy, and should expect to rewrite our software frequently. Program restructuring tools make it easier to rewrite software, and so should be a key part of every software development environment.

Although program restructuring tools have never had high visibility, they have been around a long time. Some of the first were used to convert programs based on goto statements to use structured control statements and to help convert 1401 assembly language programs to COBOL. Some program restructuring systems eliminate argument over code formatting standards by allowing each programmer to view the software using his or her own standard. More recently, there have been programs to convert non-object-oriented C or C++ programs to object-oriented C++ programs, programs to convert from one version of a language to the next, and programs to eliminate unneeded ifdefs from C programs. Most of these tools were not interactive.

My experience is with the Refactoring Browser, an open source program restructuring tool for Smalltalk (http://wiki.cs.uiuc.edu/RefactroingBrowser). This tool makes it easy to rename, move, split, and delete program elements safely. It can quickly (in a second or less) tell whether an action is safe, and will warn about unsafe actions. Safe actions are guaranteed to not change the behavior of the program. The Refactoring Browser is part of an interactive development environment, and making it fast, safe, and easy to use was more important to us than making it powerful. Many prominent members of the Smalltalk community picked it up right away and started promoting it heavily. Even so, it took about 3 or 4 years before it was widely used. Currently it is supported by 4 of the leading Smalltalk environments, which is nearly half the environments, but the ones used by 90% of the Smalltalk programmers. The programmers who use it hate to be without it, but most Smalltalk programmers still do not use it.

Refactoring (another name for restructuring that is used in some object-oriented circles) has been brought to prominence by the promoters of Extreme Programming (XP). It is explained quite well in the book "Refactoring" by Martin Fowler (http://www.refactoring.com). The idea is to constantly improve your program by making small, behavior-preserving changes to it. This enforces coding standards, reduces duplication and the size of the program, improves modularity, and causes new abstractions to emerge from the code. The book is primarily aimed at new code and at existing systems that are being actively extended, but it can also be used for legacy systems. With the rise of interest in refactoring, several companies are now selling refactoring tools for Java.

Refactoring does not require tools, though tools are very helpful. It depends more on other practices, such as building automated unit tests and frequent code reviews or, as XP teaches, pair programming. Mostly it depends on programmers being taught to tell good code from bad code and being told to fix bad code whenever they see it. Programmers also have to be taught to balance time refactoring with time spent adding new features. XP practitioners refactor even without tools, but the lack of tools is probably one reason why they report that refactoring is one of their practices that is most often omitted. Refactoring tools can cut the time to carry out refactorings by 90%, which dramatically shifts the cost/benefit curve in favor of refactoring.

Program restructuring is becoming more important as the rate of technological change continues to grow. Technological change has two effects; old systems need to be changed to use it, and most programmers will be using the technology for the first time, so will misuse it. For example, object-oriented programming features get added to languages like Ada or Visual Basic, but the old code doesn't use them, and the old programmers don't use them, either. Often one part of a system has a different style than another, based solely on when it was written, or by whom. Without massive program restructuring, the architecture of the system decays and the system becomes harder to understand. Eventually it collapses under the weight of all the changes that have been made. Program restructuring can create or recreate a good architecture, but it is usually perceived as too expensive, and sometimes it is. Program restructuring tools can make it cheaper, but it is just as important for developers to understand how to use program restructuring.

Program restructuring will be very important in the future. We will continue to collect legacy systems, and we will have to convert them to the latest standards. We will probably continue to create new standards faster and faster. Even though these new standards will make our software more flexible and easier to change, it will take work to convert to them.

For example, model driven software is flexible and easy to change. But it is hard to know what your models should be at the start. On one project, we realized in the middle of the project that a lot of information that was embedded in the software should instead be in the database. First, we refactored the code so that it produced metadata that described its function and then passed the metadata to an interpreter that carried out its function. We tested the code to make sure it behaved as it had before. Then we changed the code to write the metadata to a database, and had the interpreter read its commands from the database. Once that worked, we deleted the code that produced the metadata and just depended on the database. In this way, we reverse engineered the metadata from the code. However, we didn't think of what we were doing as reverse engineering, we were just reducing the complexity of our system, and making it easier to add the next round of features. So, program restructuring tools can make it easier to discover the proper models for a system and to convert to them.

Iterative development also increases the demands for program restructuring. Iterative development assumes that mistakes will be made, and tries to make it easier to detect

them so they can be corrected early. Some of these mistakes will be local, and can be fixed with little program restructuring. But others are global, and require changes to all parts of the system. In the past, these kinds of errors were ignored because they were considered too expensive to fix. But iterative development will provide the time to fix them if we have the will. Program restructuring tools can reduce the cost further.

Program restructuring has been around for a long time, and has not revolutionized the world. In fact, most programmers have never used a program restructuring tool, or even know that they exist. Why is there so little success? I think the reasons are technical, social, and economic.

1) Program restructuring is language dependent. A tool for C will not work for FORTRAN. Some restructuring tools have a language independent program representation and can handle several different languages (such as DMS from Semantic Designs), but this adds to the complexity of the tool. As languages change, the tools have to change, and this adds to their cost. Further, many projects use several languages, and this makes it harder for them to find useful restructuring tools.

2) Program restructuring tools should be integrated into the IDE. In our experience, restructuring must be done interactively and in an exploratory environment. It is hard to tell whether a transformation will be an improvement until you see its effects. Often you have to apply several transformations before you can be sure that the original one is an improvement. It is important to be able to quickly see the results of a transformation and to be able to undo changes that turn out to be a mistake. But because there are so many IDEs, integrating a tool into an IDE will limit its market.

3) There are still many unsolved technical restructuring problems. For example, there is no complete solution for handling C preprocessor directives. There will never be a complete solution for analyzing pointers, since alias detection is undecidable, but there are probably better solutions than the ones we have now. Unsolved technical problems are not the main reason keeping restructuring tools from being more popular, but they are still problems that should be solved.

4) Software developers do not know how to restructure programs. They think in terms of avoiding restructuring, not how to make best use of it. They think of restructuring as part of software reengineering, not as a part of normal development. Their software development processes and their software development environments do not provide a place for restructuring, and so make it difficult.

5) There is no incentive for tool-makers to make program restructuring tools. It is hard enough to make money from compilers, even though every programmer has to use them. It is much harder to make money from tools that nobody thinks they

need. Until software developers start demanding restructuring tools, it is unrealistic to expect tool vendors to provide them.

6) In spite of the history of program restructuring, there hasn't been much research on it. Most program restructuring tools have been proprietary and expensive. There are no conferences where tool builders can talk to each other and to their users. There is not really a program restructuring community.

These problems can be solved. The tools are not difficult to create, no harder than compilers. If customers demanded them then tool-makers would provide them. But people won't demand them if they don't know what they are. If we develop good tools for one environment, people will start to demand them for other environments. As knowledge expands, demand will grow, which will lead to more tools, more experience, and thus more demand.

An obvious way to stimulate use of program restructuring is to fund the development of program restructuring tools. But this is not sufficient for success, and might not be necessary. It is much more important to apply program restructuring to real projects to learn how program restructuring should change the software development process and to learn the characteristics of successful program restructuring tools.

A great way to stimulate progress in program restructuring is to pick some systems that need to be restructured and to restructure them. The best projects are ones that need to be extended, but their design is so complex that changing them seems impossible. I've worked on a couple of systems like this and so I know that restructuring them can work wonders. Giving these systems a good structure can revive them and give them many more years of useful life. Although tool venders will be funded to create new tools or improve old ones, even more funding should be given to the actual restructuring effort. The restructuring should be led by researchers who plan to write about their experiences. They should reflect on the processes they are following, and constantly try to improve them. They should criticize the tools they use and try to improve them. They should share experiences and learn from other projects.

Restructuring is important because software is an asset. Assets like buildings have to be maintained and renovated to keep their value high. Software also has to be maintained and renovated to keep its value high. We need to learn how to take care of software better, and we need better tools and practices for taking care of it.